Universidad de Oviedo
Escuela Politécnica de Ingeniería de Gijón
Departamento de Informática
GIJÓN

Prácticas de
Sistemas de Información

Identificación de procesos de negocio

Nombre corto: PNJU_COIIPA
Código del equipo: 2022G41

Doc. Id.: PCPA_00
Versión: 1.0

Fecha: 2023-05-11

Escrito por:
- Rodríguez López, Alejandro
- Mier Montoto, Juan
- Revilla Hernández, Manuel
- Martínez Ginzo, Rubén

Carácter: Definitivo

Equipo de trabajo

EDS	Rodríguez López, Alejandro
EDS	Mier Montoto, Juan
EDS	Revilla Hernández, Manuel
EDS	Martínez Ginzo, Rubén

Tabla de Contenidos

Historia

Versión	Fecha	Cambios introducidos
0.1	26/04/2023	Estructura y redacción de la introducción. "Identificación" de los procesos de negocio.
0.2	09/05/2023	Añadidos primeros borradores de las pruebas.
1.0	11/05/2023	Refinamiento final de las pruebas realizadas.

1. Introducción

Se redacta este documento con el objetivo de identificar y describir los procesos de negocio del sistema que puedan ser probados sin interfaz de usuario. También se llevará a cabo el diseño de las pruebas con JUnit, así como su implementación.

Con la realización de estas pruebas, se pretende asegurar la máxima calidad en el software desarrollado por este equipo.

El sistema se ha diseñado para la gestión del COIIPA, el Colegio Oficial de Ingenieros en Informática del Principado de Asturias. Las funcionalidades claves de la aplicación desarrollada son: la administración y creación de cursos, la gestión de inscripciones de alumnos, el control de pagos de inscripciones y la gestión de pagos y facturas a profesores.

2. Procesos de negocio identificados

Para identificar los distintos procesos de negocio, se ha desglosado el proyecto en diferentes ámbitos, y a continuación, estos ámbitos se han desgranado en diferentes procesos de negocio. A continuación, se muestran los procesos identificados, de los cuáles, una selección de ellos, serán probados mediante pruebas unitarias realizadas con *JUnit*.

1. Cursos
    1. Registrar cursos
        1. Con profesores a cargo del COIIPA
        2. Con profesores a cargo de una entidad externa
    2. Modificar cursos
    3. Cancelación de cursos
2. Inscripciones
    1. Inscripciones unitarias
    2. Inscripciones múltiples
    3. Cancelación
3. Pagos de alumnos
4. Facturas
    1. A profesores
    2. A empresas

3. Asignación de los procesos de negocio

A continuación, se definirán las responsabilidades de cada miembro del equipo a la hora de realizar las pruebas de los procesos de negocio:

- Rodríguez López, Alejandro: Planificación de cursos y sesiones, Facturas.
- Revilla Hernández, Manuel: Inscripciones de alumnos, Pagos de alumnos.
- Mier Montoto, Juan Francisco: Inscripciones múltiples, Modificar cursos.
- Martínez Ginzo, Rubén: Cancelación de un curso, Cancelaciones de inscripciones.

4. Implementación de las pruebas

4.1. Introducción a las pruebas

A continuación, se presentan las distintas pruebas realizadas con el objetivo de hallar errores en el software desarrollado hasta el momento. Para cada prueba, se indica la siguiente información (si se aplica):

1. El estado inicial de la base de datos.
2. La función a ejecutar, así como una breve descripción de lo que debería hacer.
3. Cualquier efecto secundario que deba tener la función.
4. Los parámetros que son pasados a dicha función.
5. El estado final de la base de datos.
6. El estado final del programa.

4.2. Planificación de Cursos y Sesiones

Objetivo: El responsable de formación busca registrar un nuevo curso.

Descripción:

1. Se le asigna nombre y descripción al curso.
2. Se le asignan fechas de inicio y final al período de inscripción del curso.
    a. La fecha de fin del período de inscripción no puede ser anterior a la de inicio.
3. Se le asignan fechas de inicio y final al período lectivo del curso.
    a. La fecha de fin del período lectivo no puede ser anterior a la de inicio.
    b. La fecha de inicio del período lectivo puede ser anterior a la de fin del período de inscripción.
    c. La fecha de inicio del período lectivo no puede ser anterior a la de inicio del período de inscripción.
4. Se le asigna el número máximo de plazas al curso.
5. Se le asigna el coste que deberá pagar cada colectivo para inscribirse en el curso.
    a. El coste a abonar debe ser estrictamente superior a 0.
    b. Debe existir al menos un par colectivo-coste.
6. Se introducen las sesiones en las que se impartirá el curso.
    a. Una sesión consta de:
        i. Localización
        ii. Fecha (perteneciente al intervalo comprendido entre el inicio del período lectivo y el final del mismo).
        iii. Hora de inicio
        iv. Hora de fin
7. Se selecciona uno o varios profesores para impartir el curso.
    a. Al insertar una remuneración a un profesor, éste queda seleccionado para impartir el curso.
    b. La remuneración debe ser estrictamente superior a 0.
8. Un curso también puede ser impartido por profesores de una entidad externa, en este caso, en lugar de asignar remuneración a uno o varios profesores, se le asigna una cantidad a la entidad seleccionada.

4.2.1. Pruebas de Cursos

El registro de un curso es independiente de la asignación de profesores y sesiones al mismo; por ello, estas tres acciones se verán por separado.

1. Registrar curso
2. Registrar sesiones a un curso
3. Asignar profesores a un curso

4.2.1.1. Registrar curso

4.2.1.1.1. Estado inicial de la BBDD

No es necesario que la BBDD tenga contenido alguno para crear un curso. Para las siguientes pruebas (hasta nuevo aviso), la BBDD está vacía.

4.2.1.1.2. Función a ejecutar

En las siguientes pruebas se llamará a la función **`insertCurso(String nombre, String descripcion, String startInscr, String endInscr, String start, String end, String plazas, Map<String, Double> costes).`** La función recibe toda la información de un curso, desde su nombre hasta las fechas que lo limitan, así como el número total de plazas y un diccionario que contenga los nombres de los distintos colectivos que se pueden inscribir junto a los precios que deberán abonar.

La función **`insertCurso`** retornará un dato **`ReturnValue<String>`**, que describe la salida de la función. **`ReturnValue`** tiene un atributo **`boolean`** que será **`false`** cuando haya resultado un error y **`true`** cuando haya resultado sin errores. También tiene un atributo de tipo **`T`** que puede corresponder al tipo de dato que retorne la función.

4.2.1.1.3. Pruebas realizadas

En todas las pruebas se han utilizado los mismos datos, haciendo pequeñas variaciones para forzar fallos. Para no listar toda la información repetidamente, se listan aquí los datos por defecto (correctos) y en cada prueba se anotará únicamente el dato modificado.

```
private RegistrarCursoModel rcm;
private String nombre = "Kafka";
private String descripcion = "En este curso se estudiarán las bases de Kafka.";
private String startInscr = "2023-01-01";
private String endInscr = "2023-02-01";
private String start = "2023-02-02";
private String end = "2023-05-01";
private String plazas = "20";
private Map<String, Double> costes = new HashMap<String, Double> () {
private static final long serialVersionUID = 1L;
{
	this.put("Colegiados", 30.0);
	this.put("Estudiantes de Uniovi", 10.0);
}};
```

4.2.1.1.3.1. Curso correcto

```
@Test
/**
 * This test inserts a Curso that is all okay.
 * The method will return the id of the Curso.
 * As we have just created the DB, this ID will be 1.
 */
public void test1() {
	assertEquals(
		"1",
		rcm.insertCurso(
			this.nombre,
			this.descripcion,
			this.startInscr, this.endInscr, this.start, this.end,
			this.plazas,
			this.costes
		).get()
	);
}
```

4.2.1.1.3.2. Costes nulos

```
@Test
/**
 * This test inserts a Curso with null costs for the different collectives.
 * This will return an error in the form of {@code false} in the {@link ReturnValue}.
 * @see ReturnValue
 */
public void test2() {
	assertEquals(
		false,
		rcm.insertCurso(
			this.nombre,
			this.descripcion,
			this.startInscr, this.endInscr, this.start, this.end,
			this.plazas,
			null
		).isOkay()
	);
}
```

4.2.1.1.3.3. Sin costes

```
@Test
/**
 * This test inserts a Curso with empty costs for the different collectives.
 * This will return an error in the form of {@code false} in the {@link ReturnValue}.
 * @see ReturnValue
 */
public void test3() {
	assertEquals(
		false,
		rcm.insertCurso(
			this.nombre,
			this.descripcion,
			this.startInscr, this.endInscr, this.start, this.end,
			this.plazas,
			new HashMap<String, Double> ()
		).isOkay()
	);
}
```

4.2.1.1.3.4. –5 plazas

```
@Test
/**
 * This test inserts a Curso with negative free seats.
 * This will return an error in the form of {@code false} in the {@link ReturnValue}.
 * @see ReturnValue
 */
public void test4() {
	assertEquals(
		false,
		rcm.insertCurso(
			this.nombre,
			this.descripcion,
			this.startInscr, this.endInscr, this.start, this.end,
			"-5",
			this.costes
		).isOkay()
	);
}
```

4.2.1.1.3.5. 0 plazas

```
@Test
/**
 * This test inserts a Curso with no free seats.
 * This will return an error in the form of {@code false} in the {@link ReturnValue}.
 * @see ReturnValue
 */
public void test5() {
	assertEquals(
		false,
		rcm.insertCurso(
			this.nombre,
			this.descripcion,
			this.startInscr, this.endInscr, this.start, this.end,
			"0",
			this.costes
		).isOkay()
	);
}
```

4.2.1.1.3.6. Fecha de inicio de inscripción posterior al fin del curso

```
@Test
/**
 * This test inserts a Curso whose start_inscr date is after the end of the Curso.
 * This will return an error in the form of {@code false} in the {@link ReturnValue}.
 * @see ReturnValue
 */
public void test6() {
	assertEquals(
		false,
		rcm.insertCurso(
			this.nombre,
			this.descripcion,
			"2025-01-01", "2023-02-01", "2023-02-02", "2023-05-01",
			this.plazas,
			this.costes
		).isOkay()
	);
}
```

4.2.1.1.3.7. Fecha de inicio de inscripción posterior al fin de la fecha de inscripciones

```
@Test
/**
 * This test inserts a Curso whose start_inscr date is after the end of the end_inscr of the
Curso.
 * This will return an error in the form of {@code false} in the {@link ReturnValue}.
 * @see ReturnValue
 */
public void test7() {
assertEquals(
		false,
		rcm.insertCurso(
			this.nombre,
			this.descripcion,
			"2023-01-01", "2022-02-01", "2023-02-02", "2023-05-01",
			this.plazas,
			this.costes
		).isOkay()
	);
}
```

4.2.1.1.3.8. Fecha de inicio anterior al inicio de inscripciones

```
@Test
/**
 * This test inserts a Curso whose start date is before the start_inscr of the Curso.
 * This will return an error in the form of {@code false} in the {@link ReturnValue}.
 * @see ReturnValue
 */
public void test8() {
	assertEquals(
		false,
		rcm.insertCurso(
			this.nombre,
			this.descripcion,
			"2023-01-01", "2023-02-01", "2022-02-02", "2023-05-01",
			this.plazas,
			this.costes
		).isOkay()
	);
}
```

4.2.1.1.3.9. Curso correcto asignado a una entidad externa

```
@Test
/**
 * This test inserts a Curso assigned to an external entity.
 * There are no errors in this insertion, so the method returns the new ID ("1").
 */
public void test10 () {
	this.getDatabase().executeUpdate(
		"INSERT INTO entidad "
		+ "(nombre, email, telefono) VALUES "
		+ "('cursados', 'info@cursados.com', '775221330')");
	assertEquals(
		"1",
		rcm.insertCursoExterno(
			this.nombre,
			this.descripcion,
			"2023-01-01", "2023-02-01", "2023-02-02", "2023-05-01",
			this.plazas,
			this.costes,
			"1",
			"500"
		).get()
	);}
```

4.2.1.1.3.10. Pago negativo

```
@Test
/**
 * This test inserts a Curso assigned to an external entity.
 * There is an error in the payment assigned to this entity, as it is negative.
 * The method will return an error as false.
 */
public void test11 () {
	this.getDatabase().executeUpdate(
		"INSERT INTO entidad "
		+ "(nombre, email, telefono) VALUES "
		+ "('cursados', 'info@cursados.com', '775221330')");
		assertEquals(
			false,
			rcm.insertCursoExterno(
				this.nombre,
				this.descripcion,
				"2023-01-01", "2023-02-01", "2023-02-02", "2023-05-01",
				this.plazas,
				this.costes,
				"1",
				"-500"
		).isOkay()
	);
}
```

4.2.1.1.3.11. Pago 0

```
@Test
/**
 * This test inserts a Curso assigned to an external entity.
 * There is an error in the payment assigned to this entity, as it is zero.
 * The method will return an error as false.
 */
public void test12 () {
	this.getDatabase().executeUpdate(
		"INSERT INTO entidad "
		+ "(nombre, email, telefono) VALUES "
		+ "('cursados', 'info@cursados.com', '775221330')");
	assertEquals(
		false,
		rcm.insertCursoExterno(
			this.nombre,
			this.descripcion,
			"2023-01-01", "2023-02-01", "2023-02-02", "2023-05-01",
			this.plazas,
			this.costes,
			"1",
			"0"
		).isOkay()
	);
}
```

4.2.1.2. Registrar sesiones a un curso

4.2.1.2.1. Estado inicial de BBDD

La BBDD contiene un curso:

```
private RegistrarCursoModel rcm;
private String nombre = "Kafka";
private String descripcion = "En este curso se estudiarán las bases de Kafka.";
private String startInscr = "2023-01-01";
private String endInscr = "2023-02-01";
private String start = "2023-02-02";
private String end = "2023-05-01";
private String plazas = "20";
private Map<String, Double> costes = new HashMap<String, Double> () {
private static final long serialVersionUID = 1L;
{
	this.put("Colegiados", 30.0);
	this.put("Estudiantes de Uniovi", 10.0);
}};
```

4.2.1.2.2. Función a ejecutar

En las siguientes pruebas se llamará a la función **`insertEvento(List<SesionDTO> eventos, String idCurso)`**. Como se puede ver, la función admite insertar varios eventos simultáneamente. De los eventos pasados como parámetro, eliminará aquellos que no sean válidos y no los insertará. Retorna un **`ReturnValue<Integer>`**, que contiene un **`Integer`** que corresponde a la cantidad de sesiones que han sido insertadas en la BBDD.

4.2.1.2.3. Pruebas realizadas

4.2.1.2.3.1. Evento correcto

```
@Test
/**
 * This test inserts a new Evento to a Curso.
 * There is no error on the Evento as its date matches the allowed window.
 */
public void test1 () {
	assertEquals(Integer.valueOf(1), rcm.insertEvento(
		new java.util.ArrayList<SesionDTO> () {{
			this.add(new SesionDTO(
				"AN-B3",
				"2023-02-03",
				"09:00",
				"10:00"
			));
		}},
		this.cursoId
	).get());
}
```

4.2.1.2.3.2. Fecha anterior al inicio del curso

```
@Test
/**
 * This test inserts a new Evento to a Curso.
 * This Evento takes place before the Curso starts, so it's denied and nothing is added.
 * The function returns that 0 Eventos have been added.
 */
public void test2 () {
	assertEquals(Integer.valueOf(0), rcm.insertEvento(
		new java.util.ArrayList<SesionDTO> () {{
			this.add(new SesionDTO(
				"AN-B3",
				"2023-01-01",
				"09:00",
				"10:00"
			));
		}},
		this.cursoId
	).get());
}
```

4.2.1.2.3.3. Varias sesiones simultáneamente

```
@Test
/**
 * This test inserts a new Evento to a Curso.
 * In this test, several Eventos will be added (4), two of which contain mistaked in their dates.
 * Therefore, two are added and two are denied by the function.
 */
public void test3 () {
	assertEquals(Integer.valueOf(2), rcm.insertEvento(
		new java.util.ArrayList<SesionDTO> () {{
			this.add(new SesionDTO(
				"AN-B3",
				"2023-02-07", // Ok (allowed)
				"09:00",
				"10:00"
			));
			this.add(new SesionDTO(
				"AN-B3",
				"2023-04-05", // Ok (allowed)
				"09:00",
				"10:00"
			));
			this.add(new SesionDTO(
				"AN-B3",
				"2020-01-01", // Way too soon (denied)
				"09:00",
				"10:00"
			));
			this.add(new SesionDTO(
				"AN-B3",
				"2025-01-01", // Way too late (denied)
				"09:00",
				"10:00"
			));
		}},
		this.cursoId
	).get());
}
```

4.2.1.3. Registrar sesiones a un curso

4.2.1.3.1. Estado inicial de BBDD

La BBDD a utilizar contiene un curso, que en este caso será el utilizado para el apartado anterior, véase 4.2.1.2.1.

La BBDD también contiene un par de profesores:

```
this.profesores = new java.util.ArrayList<ProfesorDTO> () {
	private static final long serialVersionUID = 1L;
{
add(new ProfesorDTO("Pedro", "Angel Martinez", "545232121A", "C/ Albeniz, 5", "panma@gmail.com",
"664225423"));
add(new ProfesorDTO("Miguel", "Rubio Sanz", "789776221C", "Avd. Sol, 22", "miru@gmail.com",
"123222888"));
}
};
```

4.2.1.3.2. Función a ejecutar

En las siguientes pruebas se llamará a la función **`insertDocencia(List<ProfesorDTO> profesores, String cursoId)`**. Como se puede ver, la función admite insertar varios eventos simultáneamente. De los profesores pasados como parámetro, eliminará aquellos que no sean válidos y no los insertará. Retorna un **`ReturnValue<Integer>`**, que contiene un **`Integer`** que corresponde a la cantidad de profesores que han sido insertados en la BBDD.

Al tratarse de inserciones en una tabla intermedia, no se están insertando profesores realmente, sino que se están asociando sus ID con los ID de cada curso y sus correspondientes remuneraciones.

4.2.1.3.3. Pruebas realizadas

4.2.1.3.3.1.1. Profesor correcto

```
@Test
/**
 * This test inserts a new Profesor to a Docencia.
 * In this test, only one Profesor is inserted with correct data, so no error is returned.
 */
public void test1 () {
	profesores.get(0).setRemuneracion("250");
	assertEquals(
		Integer.valueOf(1),
		this.rcm.insertDocencia(profesores.subList(0, 1), cursoId).get()
	);
}
```

4.2.1.3.3.1.2 Profesor incorrecto

```
@Test
/**
 * This test inserts a new Profesor to a Docencia.
 * In this test, only one Profesor is inserted with incorrect data, so none are inserted.
 */
public void test10 () {
	profesores.get(0).setRemuneracion("-250");
	assertEquals(
		Integer.valueOf(0),
		this.rcm.insertDocencia(profesores.subList(0, 1), cursoId).get()
	);
}
```

4.2.1.3.3.2. Varios profesores

4.2.1.3.3.2.1 Ambos correctos

```
@Test
/**
 * This test inserts a new Profesor to a Docencia.
 * In this test, two Profesores are inserted, both of which contain correct data, so no error is
returned.
 */
public void test2 () {
	profesores.get(0).setRemuneracion("250");
	profesores.get(1).setRemuneracion("200");
	assertEquals(
		Integer.valueOf(2),
		this.rcm.insertDocencia(profesores, cursoId).get()
	);
}
```

4.2.1.3.3.2.2. Varios profesores, uno incorrecto

```
@Test
/**
 * This test inserts a new Profesor to a Docencia.
 * In this test, two Profesores are inserted, only one of which contains correct data, so only
one is inserted.
 */
public void test3 () {
	profesores.get(0).setRemuneracion("250");
	profesores.get(1).setRemuneracion("-200");
	assertEquals(
		Integer.valueOf(1),
		this.rcm.insertDocencia(profesores, cursoId).get()
	);
}
```

4.2.1.3.3.2.3. Varios profesores, ambos incorrectos

```
@Test
/**
 * This test inserts a new Profesor to a Docencia.
 * In this test, two Profesores are inserted, both of which contain incorrect data, so none are
inserted.
 */
public void test4 () {
	profesores.get(0).setRemuneracion("-250");
	profesores.get(1).setRemuneracion("-200");
	assertEquals(
		Integer.valueOf(0),
		this.rcm.insertDocencia(profesores, cursoId).get()
	);
}
```

4.3. Cancelación de un curso

4.3.1. Descripción del proceso de negocio

1. Objetivo: cancelar un curso cuando sea necesario.

2. Descripción:

A. El responsable de formación tiene la necesidad de cancelar un curso porque no cumple los requisitos necesarios para que comience. Se deberá tener en cuenta lo siguiente:

1. El curso a cancelar.

2. El estado del curso a cancelar.

a. No se puede cancelar un curso previamente cancelado.

b. No se puede cancelar un curso que ya ha comenzado.

3. Las inscripciones relativas a dicho curso.

a. Se deberán de cancelar todas las inscripciones.

b. Si el curso no tiene inscripciones, no es necesario realizar la cancelación de las mismas.

4. El estado de estas inscripciones antes de ser canceladas.

a. Se deberá devolver el 100% del dinero pagado en inscripciones pagadas.

b. No se debe cancelar una inscripción previamente cancelada por el inscrito.

5. Una vez se cancele un curso y todas sus inscripciones, ningún alumno podrá inscribirse a dicho curso.

B. Para que un curso se cancele debe de haberse registrado anteriormente con éxito, incluyendo los datos necesarios. Si el curso no consta de ninguna inscripción, no se requiere la cancelación de las mismas.

4.3.2. Diseño de las pruebas

Para el diseño de las pruebas, se procederá a borrar todos los contenidos de la base de datos. A continuación, se registrarán varios cursos nuevos con motivo de la ejecución de las mismas.

Para la primera prueba unitaria (**testEstadoCurso**) se cubre la clase de equivalencia del estado del curso a cancelar. El estado de un curso no se almacena en la base de datos como tal, sino que se calcula durante el flujo de ejecución de la aplicación. Este cálculo se realiza en función de la fecha actual. Se realizarán los siguientes casos de prueba:

- Se insertará un curso (de forma correcta) para que la fecha actual esté contenida entre las fechas de inscripción del curso y así su estado sea *EN_INSCRIPCION*.
- Se insertará un curso para que la fecha actual sea posterior a la fecha de inscripción y esté contenida entre las fechas de inicio y fin del curso. Así su estado será *EN_CURSO*.
- Se insertará un curso con fechas de inscripción posteriores a la fecha actual, así su estado será *PLANEADO*.
- Se insertará un curso con fecha de fin anterior a la fecha inicial, así su estado será *FINALIZADO*.
- Se insertará un curso con estado CANCELADO (este es el único estado que se almacena en la base de datos).

De los casos anteriores, solo se podrá realizar la cancelación de un curso con estado *EN_INSCRIPCIÓN* o *PLANEADO*. No se puede cancelar un curso que ya ha comenzado (*EN_CURSO*) o un curso que ya haya finalizado (*FINALIZADO*). Tampoco tiene sentido cancelar un curso previamente cancelado.

Para la segunda prueba unitaria (**testInscripcionesCurso)** se cubre la clase de equivalencia de cancelación de inscripciones de un curso. Para ello se insertarán varias inscripciones en los cursos con estado *EN_INSCRIPCIÓN* y *PLANEADO* (los cursos que sí pueden cancelarse). El estado de las inscripciones se calcula en función de los pagos realizados.

- Se insertará una inscripción con estado *PAGADA* en el curso con estado *EN_INSCRIPCIÓN*.
- Se insertará una inscripción con estado *PENDIENTE* en el curso con estado *EN_INSCRIPCIÓN*.
- Se insertará una inscripción con estado *CANCELADA* en el curso con estado *EN_INSCRIPCIÓN*.
- Se insertarán los respectivos pagos necesarios para que el estado de la primera inscripción insertada sea *PAGADA*.

Al curso con estado *PLANEADO* no se le puede añadir ninguna inscripción, ya que es un curso para el que no ha comenzado el periodo de inscripción. Para este curso no se debería de realizar el proceso de cancelación de inscripciones.

En el caso del curso anterior (estado *EN_INSCRIPCIÓN*), cuando se cancela el mismo se deben de cancelar las inscripciones con estado *PAGADA* y *PENDIENTE*, devolviendo el importe pagado a la inscripción con estado *PAGADA*.

4.3.3. Implementación de las pruebas en JUnit

```
package g41.si2022.coiipa.gestionar_curso;

import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;

import java.time.LocalDate;
import java.util.List;

import org.junit.*;

import g41.si2022.dto.CursoDTO;
import g41.si2022.dto.InscripcionDTO;

public class GestionarCursoTest extends g41.si2022.coiipa.TestCase {

    private static GestionarCursoModel model;

    @Override
    public void loadData() {
        try {

this.getDatabase().executeScript("src/test/java/g41/si2022/coiipa/gestionar_curso/datos.sql");
        } catch (Exception e) {
            System.exit(1);
        }
    }

    @BeforeClass
    public static void initialize() {
        model = new GestionarCursoModel();
    }

    private CursoDTO getCursoDTO(String id) {
        // Obtener curso en función de la id
        CursoDTO curso = model.getCurso(id);
        // Calcular estado del curso
        curso.updateEstado(LocalDate.now());

        return curso;
    }

    private List<InscripcionDTO> getInscrCurso(String idCurso) {
        // Obtener inscripciones del curso pasado por parámetro
        List<InscripcionDTO> inscripciones = model.getCursoInscripciones(idCurso);
        // Calcular estado de las isncripciones
        for (InscripcionDTO inscripcion: inscripciones)
            inscripcion.updateEstado(LocalDate.now());

        return inscripciones;
    }
```

```java
/**
 * Controlar el estado de un curso para su posterior cancelación.
 */
@Test
public void testEstadoCurso() {
    CursoDTO curso;
    /*
     * Sólo se puede cancelar un curso con estado EN_INSCRIPCION o PLANEADO.
     * Los datos introducidos para este test son:
     *     - Curso con estado EN_INSCRIPCION (id:1)
     *     - Curso con estado EN_CURSO (id:2)
     *     - Curso con estado PLANEADO (id:3)
     *     - Curso con estado FINALIZADO (id:4)
     *     - Curso con estado CANCELADO (id:5)
     */

    // Curso con estado EN_INSCRIPCION
    curso = getCursoDTO("1");
    // Debería devolver cierto
    assertTrue(model.cancelarCursoTest(curso));

    // Curso con estado EN_CURSO
    curso = getCursoDTO("2");
    // Debería devolver falso
    assertFalse(model.cancelarCursoTest(curso));

    // Curso con estado PLANEADO
    curso = getCursoDTO("3");
    // Debería devolver cierto
    assertTrue(model.cancelarCursoTest(curso));

    // Curso con estado FINALIZADO
    // Debería devolver falso
    curso = getCursoDTO("4");
    assertFalse(model.cancelarCursoTest(curso));

    // Curso con estado CANCELADO
    // Debería devolver falso
    curso = getCursoDTO("5");
    assertFalse(model.cancelarCursoTest(curso));
}
```

```java
@Test
    public void testInscripciones() {
        // Obtener curso con estado EN_INSCRIPCION
        CursoDTO curso_EN_INSCRIPCION = getCursoDTO("1");
        // Obtener curso con estado PLANEADO
        CursoDTO curso_PLANEADO = getCursoDTO("2");
        List<InscripcionDTO> inscripciones;

        /*
         * Se probará la cancelación de inscripciones de los cursos que se pueden cancelar
         * (EN_INSCRIPCIÓN y PLANEADO).
         *
         * Para este test se insertan los siguientes datos:
         *     - Inscripción con estado PAGADA (id:1) del curso EN_INSCRIPCION (id:1)
         *     - Inscripción con estado PENDIENTE (id:2) del curso EN_INSCRIPCION (id:1)
         *     - Inscripción con estado CANCELADA (id:3) del curso EN_INSCRIPCION (id:1)
         *
         * El curso en estado PLANEADO no puede tener inscripciones, ya que aún no ha
         * comenzado el periodo de inscripción. Por lo tanto, no tiene inscripciones para
         * cancelar.
         */

        // Comprobar proceso de cancelación de inscripciones para el curso EN_INSCRIPCIÓN,
        // que tiene tres inscripciones, debe devolver cierto
        assertTrue(model.cancelarInscrCursoTest(curso_EN_INSCRIPCION));

        // Comprobar proceso de cancelación de inscripciones para el curso PLANEADO, no tiene
        // inscripciones, debe devolver falso
        assertFalse(model.cancelarInscrCursoTest(curso_PLANEADO));


        // Obtener inscripciones del curso en estado EN_INSCRIPCION
        inscripciones = getInscrCurso("1");
        InscripcionDTO inscripcion1 = inscripciones.get(0);
        InscripcionDTO inscripcion2 = inscripciones.get(1);
        InscripcionDTO inscripcion3 = inscripciones.get(2);

        // Debe devolver cierto, se imprime un mensaje indicando que se tiene que realizar
        // una devolución
        assertTrue(model.cancelarInscripcionTest(inscripcion1));

        // Debe devolver cierto, se indica que no se debe realizar devolución
        assertTrue(model.cancelarInscripcionTest(inscripcion2));

        // Debe devolver falso
        assertFalse(model.cancelarInscripcionTest(inscripcion3));
    }
}
```

4.4. Cancelación de inscripciones

4.4.1 Descripción del proceso de negocio

1. Objetivo: cancelar la inscripción de un inscrito a un curso.

2. Descripción:

A. El inscrito solicita la cancelación de su inscripción.

1. No se puede solicitar la cancelación de una inscripción con estado cancelado.

2. Debe controlarse que los datos del solicitante se corresponden con los almacenados en la respectiva inscripción.

B. Debe de controlarse el estado del curso al que está inscrito.

1. No se puede cancelar la inscripción de un curso que ya ha comenzado.

2. Sólo se pueden cancelar inscripciones con un curso en estado *EN_INSCRIPCION*.

4.4.2. Diseño de las pruebas

Para el diseño de las pruebas se borrarán todos los datos almacenados en la base de datos y se almacenarán los necesarios para la ejecución de las mismas.

En la primera prueba unitaria (**testEstadoCancelarInscripcion**) se cubre la clase de equivalencia de la cancelación de la inscripción. Cuando se solicita la cancelación de una inscripción debe de comprobarse previamente su estado. Se insertarán diferentes inscripciones:

- Una inscripción con estado *PENDIENTE*, para la que se solicita su cancelación.
- Una inscripción con estado *CANCELADA*, que ya ha sido cancelada previamente.
- *No se insertan más inscripciones para esta prueba con el resto de los estados (*EXCESO, PAGADA, RETRASADA, RETRASADA_EXCESO, EN_ESPERA*) ya que resulta redundante. En estos casos la inscripción debe de cancelarse.
- También se insertan los datos de los alumnos que previamente han realizado sus inscripciones.

Para los casos anteriores, se comprueba que la inscripción con estado *PENDIENTE* se cancela, mientras que la inscripción con estado *CANCELADA* no.

En la segunda prueba unitaria (**testEstadoCursoCancelarInscripcion**) se cubre la clase de equivalencia del estado del curso correspondiente a la inscripción a cancelar. Durante la cancelación de una inscripción debe controlarse el estado del curso correspondiente. Para ello se insertarán nuevos cursos:

- Un curso con estado *EN_INSCRIPCION*, al que pertenece la anterior inscripción a cancelar (la inscripción con estado *PENDIENTE*).
- Una nueva inscripción, con estado *PENDIENTE* también, que debe cancelarse y corresponde al siguiente curso con estado *EN_CURSO*.
- Un curso en estado *EN_CURSO*, al que pertenece la inscripción anterior.

Para los casos anteriores, la inscripción correspondiente al curso con estado *EN_INSCRIPCIÓN* debería de poder cancelarse, mientras que la inscripción correspondiente al curso con estado *EN_CURSO* no se puede cancelar.

4.4.3. Implementación de las pruebas en JUnit

```
package g41.si2022.coiipa.gestionar_curso;

import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;

import java.beans.Transient;
import java.time.LocalDate;
import java.util.List;

import org.junit.*;

import com.fasterxml.jackson.databind.jsontype.impl.AsExistingPropertyTypeSerializer;

import g41.si2022.coiipa.gestionar_inscripciones.GestionarInscripcionesModel;
import g41.si2022.dto.CursoDTO;
import g41.si2022.dto.InscripcionDTO;

public class CancelarInscripcinesTest extends g41.si2022.coiipa.TestCase {

    private static GestionarInscripcionesModel model;

    @Override
    public void loadData() {
        try {

this.getDatabase().executeScript("src/test/java/g41/si2022/coiipa/cancelar_inscripciones/datos.sq
l");
        } catch (Exception e) {
            System.exit(1);
        }
    }

    @BeforeClass
    public static void initialize() {
        model = new GestionarCursoModel();
    }

    /**
     * Método para obtener una inscripción dado su id
     * @param idInscr id de la inscripción
     * @return inscripción correspondiente a dicho id
     */
    public InscripcionDTO getInscripcion(String idInscr) {
        InscripcionDTO inscripcion = model.getInscrById(idInscr);
        inscripcion.updateEstado(LocalDate.today());

        return inscripcion;
    }

    /**
     * Método para obtener el curso mediante su id
     * @param idInscripcion id del curso
     * @return curso asociado a esa id
     */
    public CursoDTO getCursoFromInscripcion(String idCurso) {
        CursoDTO curso = model.getCursoFromInscr(idCurso);
        curso.updateEstado(LocalDate.today());

        return curso;
    }
```

```
@Test
    public void testEstadoCancelarInscripcion() {
        // Obtener la primera inscripción almacenada. Estado PENDIENTE.
        InscripcionDTO inscripcion = getInscripcion("1");

        // Se comprueba si se puede cancelar. Debe devolver cierto.
        assertTrue(model.cancelarInscripcionTest(inscripcion));

        // Obtener la segunda inscripción almacenada. Estado CANCELADA.
        inscripcion = getInscripcion("2");

        // Se comprueba si se puede cancelar. Debe devolver falso.
        assertTrue(model.cancelarInscripcionTest(inscripcion));

        /**
         * No se comprueba la cancelación para el resto de estados de una
         * inscripción ya que resulta redundante. Siempre se debe de poder
         * cancelar una inscripción exceptuando una que ya se haya cancelado.
         */
    }

    @Test
    public void testEstadoCursoCancelarInscripcion() {
        // Obtener primera inscripción almacenada a cancelar. Estado PENDIENTE.
        InscripcionDTO inscripcionACancelar1 = getInscripcion("1");

        // Obtener curso asociado a esa inscripción
        CursoDTO curso1 = getCursoFromInscripcion(inscripcionACancelar1.getCurso_id());

        // Comprobar el estado del curso para cancelar la inscripción. Se trata
        // de un curso con estado EN_INSCRIPCION, luego la inscripción se puede
        // cancelar y debe devolver cierto
        assertTrue(model.cancelarInscripcionEstadoCursoTest(curso1));

        // Obtener la tercera inscripción almacenada a cancelar. Estado PENDIENTE. Puede
        // cancelarse
        InscripcionDTO inscripcionACancelar3 = getInscripcion("3");

        // Obtener curso asociado a la inscripción 3
        CursoDTO curso2 = getCursoFromInscripcion(inscripcionACancelar3.getCurso_id());

        // Comprobar el estado del curso para cancelar la inscripción. Se trata de un curso
        // con estado EN_CURSO, luego la inscripción no se puede cancelar. Debe devolver falso
        assertFalse(model.cancelarInscripcionEstadoCursoTest(curso2));

    }
}
```

4.6. Inscripción de un nuevo alumno en el COIIPA

4.6.1 Descripción del proceso de negocio

1) Objetivo:
    a. La inscripción de un nuevo alumno en la base de datos del COIIPA, de modo que este pueda posteriormente realizar acciones con el COIIPA, como podría ser, por ejemplo, la inscripción en nuevos cursos.
2) Descripción:
    a. Un usuario desea inscribirse en el COIIPA
        i. Se le solicita el nombre, los apellidos, el email, y opcionalmente el teléfono.
    b. Se verifica que el usuario no está ya inscrito en el COIIPA.
        i. Si se supera esta verificación el proceso sigue adelante
    c. Se verifica la estructura del email
        i. Si es válida, se puede continuar con el proceso.
        ii. Si no es válida, se lanza un mensaje de error, y se cancela el proceso de inscripción.
    d. Se verifica que no hay otro usuario con ese mismo e-mail
        i. En caso de haberlo, el proceso es cancelado, lanzando un mensaje de error.
        ii. Si todo está correcto, el alumno se inscribe en la base de datos del COIIPA. Se crea una nueva entrada con sus datos.

4.6.2. Diseño de las pruebas

Para cada caso de prueba, se dispone de un método que procede a vaciar la base de datos. A continuación, se procede a insertar 3 cursos, los cuales tienen 40, 0 y 10 plazas respectivamente.

En el primer test unitario (**`testInscribirUsuarioExistenteYNoExistente)`**, se cubre la clase de equivalencia de la preexistencia del usuario. Para ellos se realizan los siguientes casos de prueba para cubrir esta clase de equivalencia.

- Se realiza una inscripción sin haber nadie inscrito previamente.
- Se realiza una inscripción, usando de nuevo los mismos datos. En este caso, el modelo nos devuelve un resultado de error, y se aborta el proceso de inscripción.

Finalmente, se puede apreciar que únicamente se ha guardado la primera inscripción, rechazándose la segunda.

En el segundo test unitario (**`testInscribirUsuarioEmailIncorrecto)`**, se cubre la clase de equivalencia sobre la validez del email.

- Se realiza una inscripción con un email mal formado. El modelo devuelve un error, y cancela el proceso.

Se puede apreciar que no se ha realizado la inscripción de ese usuario.

En el tercer test unitario (**`inscribirAlumnoConEmailYaUsado)`**, se intenta inscribir a dos usuarios distintos con un mismo email.

- Se inscribe un usuario con unos datos y un email.
- Se inscribe a otro usuario con otros datos y el mismo email. El modelo ha de rechazar esta segunda inscripción.

Finalmente, en la base de datos, nos encontramos únicamente con la inscripción original del alumno.

4.6.3. Implementación de las pruebas en Junit

```
@Test
public void testInscribirUsuarioExistenteYNoExistente() { // Prueba la condición de inscribir un
alumno que no se encuentre en la BBDD
	// Se inscribe un nuevo alumno, el cual no se encuentra inscrito aún.
	String email = "pepe@gmail.com";
	InscribirUsuarioModel modelo = new InscribirUsuarioModel();
	prepareDB(); //Para empezar el caso de prueba limpio.
	// Insertamos al usuario en la BBDD.
	assertTrue(modelo.insertAlumno("Pepe", "Pepito", email, "666666666"));
	// Devuelve verdadero, ya que el alumno aún no está.
	//Obtenemos los datos de la BBDD
	Optional<AlumnoDTO> alumno = modelo.getAlumnoFromEmail(email);
	assertEquals("Optional[AlumnoDTO(nombre=Pepe, apellidos=Pepito, email=pepe@gmail.com,
telefono=666666666, id=null, coste=null, nombreColectivo=null)]",
	alumno.toString());


    //Volvemos a intentar insertar al alumno de nuevo
    assertFalse(modelo.insertAlumno("Pepe", "Pepito", email, "666666666")); //Ahora tiene que
devolver falso.

    //Obtenemos los datos de la BBDD
   alumno2 = modelo.getAlumnoFromEmail(email);
    assertEquals("Optional[AlumnoDTO(nombre=Pepe, apellidos=Pepito, email=pepe@gmail.com,
telefono=666666666, id=null, coste=null, nombreColectivo=null)]", alumno.toString()); //El alumno
no debe aparecer inscrito 2 veces
}



@Test
public void testInscribirUsuarioEmailIncorrecto() { //Introducir un alumno con un e-mail
incorrecto
    String email = "pepelugmail.com"; //e-mail de pruebas incorrecto
    InscribirUsuarioModel modelo = new InscribirUsuarioModel();
    prepareDB(); //Para empezar el caso de prueba limpio.
    //Intentamos inscribir al usuario en la BBDD y nos debe de dar un error, ya que el email es
inscorrecto
    assertFalse(modelo.insertAlumno("Pepe", "Pepito", email, "666666666"));
    List<AlumnoDTO> listaAlumnos = modelo.getAlumnos(); //Examino la BBDD de nuevo, en busca de
las inscripciones.
    assertEquals("[]", listaAlumnos.toString());
}



@Test
public void inscribirAlumnoConEmailYaUsado() {
    String email = "pepe@gmail.com";
    InscribirUsuarioModel modelo = new InscribirUsuarioModel(); //Creo el objeto del curso
    prepareDB(); //Para empezar el caso de prueba limpio.
    assertTrue(modelo.insertAlumno("Pepe", "Pérez", email, "666666666")); //Este alumno se
inscribirá de forma correcta.
    assertFalse(modelo.insertAlumno("María", "Rodríguez", email, "666666666")); //No se podrá,
debido a que el email ya se encuentra en uso.
    List<AlumnoDTO> listaAlumnos = modelo.getAlumnos(); //Obtengo los alumnos.
```

```
    assertEquals("[AlumnoDTO(nombre=Pepe, apellidos=Perez, email=pepe@gmail.com, telefono=null,
id=null, coste=null, nombreColectivo=null)]", listaAlumnos.toString()); //Hago una consulta a la
BBDD y sólo debe haber el alumno 1.
}
```

4.7. Inscripción de un alumno en un nuevo curso del COIIPA

4.7.1. Descripción del proceso de negocio

1. Objetivo

a. La inscripción de un alumno, previamente registrado en un curso del COIIPA.

2. Descripción del proceso

a. Un alumno se desea inscribir a un curso ofrecido por el COIIPA.

1. Se verifica que dicho alumno no se encuentre ya inscrito en el curso. En caso de que ya lo esté, se cancela dicha inscripción.

2. Si se supera esta verificación, el proceso continúa adelante.

b. Se comprueba la existencia de plazas libres en dicho curso

1. Si las hay, se confirma la inscripción, y el alumno es inscrito en dicho curso

2. Si no las hay, el modelo devuelve un error, y el proceso de inscripción es cancelado.

4.7.2. Diseño de las pruebas

Para cada caso de prueba, se dispone de un método que procede a vaciar la base de datos. A continuación, procede a insertar 3 cursos, los cuales tienen 40, 0 y 10 plazas respectivamente.

En la primera prueba unitaria **`inscribirAlumnoEnCursoYaInscrito`** se cubre la clase de equivalencia para la existencia de una inscripción.

- Se inscribe al alumno en un curso. El sistema debe retornar un resultado de éxito, y permitir esta acción
- Se inscribe al mismo alumno en el mismo curso. El sistema ha de retornar un resultado de error, y rechazar esta inscripción.

Finalmente, se comprueba en la base de datos, que únicamente se ha producido la primera inscripción.

En la segunda prueba unitaria **`testInscribirAlumnoPlazasLibresYNoLibres`** se procede a inscribir al alumno en un curso con plazas libres y un curso sin ellas. Se procede de la siguiente manera:

- Se inscribe a un alumno en un curso con plazas libres. El sistema ha de retornar un resultado de éxito y permitir la inscripción de este alumno.
- A continuación, se pretende inscribir un alumno en un curso sin ninguna plaza libre. En este caso, el sistema retorna un resultado negativo, y la inscripción no se lleva a cabo.

En la base de datos, se puede apreciar que únicamente se encuentra la primera inscripción, correspondiente al curso que contiene plazas libres.

4.7.3. Implementación de las pruebas en JUnit

```
@Test
public void inscribirAlumnoEnCursoYaInscrito() { //Pruebo a inscribir un alumno dos veces en el mismo curso.

   InscribirUsuarioModel modelo = new InscribirUsuarioModel(); //Creo el objeto del curso
   prepareDB(); //Para empezar el caso de prueba limpio.

   assertTrue(modelo.insertInscripcion("2023-05-09", "4", "7", "1")); //Inscribo al alumno 7 en el curso con Id 4, que tiene 20 plazas libres. Como aún no está inscrito, debe devolver verdadero.
    assertFalse(modelo.insertInscripcion("2023-05-09", "4", "7", "1")); //Inscribo al alumno 7 en el curso con Id 4, que tiene 20 plazas libres. Como ya está inscrito, devuelve falso.
   List<InscripcionDTO> listaInscripciones = modelo.getInscripciones();
        assertEquals("[InscripcionDTO(id=null, fecha=2023-05-09, pagado=null, curso_id=4, cancelada=null, alumno_id=7, alumno_nombre=null, alumno_apellidos=null, curso_coste=null, curso_nombre=null, grupo_id=null, en_espera=null, entidad_nombre=null, estado=null)]", listaInscripciones.toString()); //Sólo hay un alumno en la tabla de inscripciones en cursos, ya que la segunda inscripción, no se ha llegado a producir.
}
```

```
@Test
public void testInscribirAlumnoPlazasLibresYNoLibres() { //Inscribo al alumno en un curso con plazas libres y otro sin ellas.
    InscribirUsuarioModel modelo = new InscribirUsuarioModel(); //Creo el objeto del curso
    prepareDB(); //Para empezar el caso de prueba limpio.
     assertTrue(modelo.insertInscripcion("2023-05-09", "1", "1", "1")); //Lo inscribo en el curso con Id 1, que tiene 40 plazas libres.
     assertFalse(modelo.insertInscripcion("2023-05-09", "2", "4", "4")); //Lo inscribo en el curso con Id 2 que tiene 0 plazas libres.
    List<InscripcionDTO> listaInscripciones = modelo.getInscripciones(); //Examino la BBDD de nuevo, en busca de las inscripciones.
        assertEquals("[InscripcionDTO(id=null, fecha=2023-05-09, pagado=null, curso_id=1, cancelada=null, alumno_id=1, alumno_nombre=null, alumno_apellidos=null, curso_coste=null, curso_nombre=null, grupo_id=null, en_espera=null, entidad_nombre=null, estado=null)]", listaInscripciones.toString());

}
```

4.8. Modificación de cursos

4.8.1. Diseño de las pruebas

4.8.1.1. Información general sobre el curso

La siguiente información general sobre un curso debe poder modificar dependiendo del estado del mismo:

- Si el curso está todavía no ha comenzado, se debe poder modificar toda la información libremente a excepción del número de plazas, que se comprobará para evitar que se inserte un número inferior al número de inscritos en el curso.
- Si el curso está en progreso, solo se deberá poder modificar la descripción - tanto el título como el número de plazas no se podrán modificar.
- Si el curso está finalizado o cerrado, no se debe poder modificar información bajo ningún concepto.

4.8.1.2. Modificación de sesiones del curso

Sobre las sesiones del curso, se deberá poder modificar lo siguiente:

- Si el curso no está en progreso todavía, se deben poder añadir y eliminar sesiones (siempre dentro de las fechas del curso). El resto de información de las sesiones no está condicionada.
- Si el curso está en progreso, se deben mantener las sesiones que ya hayan ocurrido, pero se podrán añadir nuevas sesiones.
- Si el curso está finalizado o cerrado, no se debe poder ni añadir ni eliminar sesiones bajo ningún concept

4.8.1.3. Modificación de los colectivos y costes

Sobre los colectivos y los costes de inscripción, se podrán modificar bajo las siguientes condiciones:

- Mientras no existan inscripciones asignadas al curso, se podrán modificar cualquier valor, incluyendo añadir o eliminar colectivos enteros.
- En el caso de existir inscripciones, o si el curso está marcado como “cerrado”, no se podrá modificar ningún valor.

4.8.1.4. Modificación de acuerdos con docentes

Sobre los acuerdos con los docentes o *docencias*, se debería poder modificar lo siguiente:

- Mientras el curso no esté en curso, se podrán modificar los acuerdos libremente.
- Durante el transcurso del curso, solo se podrán añadir docencias, no modificar acuerdos preexistentes.
- Una vez terminado el curso, solo se volverán a poder modificar los acuerdos a docentes, pero no eliminar ni añadir docencias.
- Una vez cerrado, no se podrá modificar ningún valor.

4.8.2. Implementación del código

```
package g41.si2022.coiipa.modificar_curso;

import java.time.LocalDate;
import java.util.LinkedList;
import java.util.List;
import java.util.stream.Collectors;

import org.junit.*;

import g41.si2022.dto.CursoDTO;
import g41.si2022.dto.DocenciaDTO;
import g41.si2022.dto.SesionDTO;


public class ModificarCursoTest extends g41.si2022.coiipa.TestCase {


private static ModificarCursosModel model;


@Override
public void loadData() {
	try
	{this.getDatabase().executeScript("src/test/java/g41/si2022/coiipa/modificar_curso/try.sql
	");}
	catch (Exception e) {
		e.printStackTrace();
		System.exit(1);
	}
}


@BeforeClass
public static void initialize() {
	model = new ModificarCursosModel();
}


private CursoDTO getDTO(String id) {
	CursoDTO curso = model.getCurso(id);
	curso.updateState(LocalDate.now());
	return curso;
}


/*
* Referencias para el test (ids de curso):
*
* - PLANEADO: 5
* - EN_INSCRIPCION: 4
* - INSCRIPCION_CERRADA: 3
* - EN_CURSO: 2
* - FINALIZADO: 1
*/
```

```java
@Test
public void testInfo() {
	CursoDTO curso;
	/*
	* La siguiente información de un curso se puede modificar dependiendo de su estado:
	*
	* - PLANEADO, EN_INSCRIPCION, INSCRIPCION_CERRADA: todo, siempre que las plazas
	* sean mayores que el número de inscritos.
	* - EN_CURSO: solo descripción.
	* - FINALIZADO, CERRADO: nada.
	*/


	// PLANEADO
	curso = getDTO("5");
	Assert.assertTrue("PLANEADO (todo ok)", model.updateCurso(curso, "Nuevo nombre", "Nueva
	descripción", "100"));
	Assert.assertFalse("PLANEADO (plazas insuficientes)", model.updateCurso(curso, "Nuevo
	nombre", "Nueva descripción", "1"));


	// EN_INSCRIPCION
	curso = getDTO("4");
	Assert.assertTrue("EN_INSCRIPCION (todo ok)", model.updateCurso(curso, "Nuevo nombre",
	"Nueva descripción", "100"));
	Assert.assertFalse("EN_INSCRIPCION (plazas insuficientes)", model.updateCurso(curso,
	"Nuevo nombre", "Nueva descripción", "1"));


	// INSCRIPCION_CERRADA
	curso = getDTO("3");
	Assert.assertTrue("INSCRIPCION_CERRADA (todo ok)", model.updateCurso(curso, "Nuevo
	nombre", "Nueva descripción", "100"));
	Assert.assertFalse("INSCRIPCION_CERRADA (plazas insuficientes)", model.updateCurso(curso,
	"Nuevo nombre", "Nueva descripción", "1"));


	// EN_CURSO
	curso = getDTO("2");
	Assert.assertTrue("EN_CURSO (descripción)", model.updateCurso(curso, curso.getNombre(),
	"Nueva descripción", curso.getPlazas()));
	Assert.assertFalse("EN_CURSO (nombre)", model.updateCurso(curso, "Nuevo nombre",
	curso.getDescripcion(), curso.getPlazas()));
	Assert.assertFalse("EN_CURSO (plazas)", model.updateCurso(curso, curso.getNombre(),
	curso.getDescripcion(), "1"));


	// FINALIZADO
	curso = getDTO("1");
	Assert.assertFalse("FINALIZADO (descripción)", model.updateCurso(curso, curso.getNombre(),
	"desc test", curso.getPlazas()));
	Assert.assertFalse("FINALIZADO (nombre)", model.updateCurso(curso, "Nuevo nombre",
	curso.getDescripcion(), curso.getPlazas()));
	Assert.assertFalse("FINALIZADO (plazas)", model.updateCurso(curso, curso.getNombre(),
	curso.getDescripcion(), "1"));
}
```

```java
@Test
public void testSesiones() {
	/*
	* Las sesiones de un curso se pueden modificar dependiendo de su estado:
	*
	* - PLANEADO, EN_INSCRIPCION, INSCRIPCION_CERRADA: todo.
	* - EN_CURSO: añadir y eliminar sesiones posteriores a la fecha de hoy.
	* - FINALIZADO, CERRADO: nada.
	*/

	CursoDTO curso;
	List<SesionDTO> sesiones;

	// PLANEADO
	curso = getDTO("5");
	sesiones = model.getSesionesFromCurso(curso.getId());
	Assert.assertTrue("PLANEADO (subset 0-1)", model.updateSesiones(curso, sesiones.subList(0,
	1), LocalDate.now()));

	// EN_INSCRIPCION
	curso = getDTO("4");
	sesiones = model.getSesionesFromCurso(curso.getId());
	Assert.assertTrue("EN_INSCRIPCION (subset 0-1)", model.updateSesiones(curso,
	sesiones.subList(0, 1), LocalDate.now()));

	// INSCRIPCION_CERRADA
	curso = getDTO("3");
	sesiones = model.getSesionesFromCurso(curso.getId());
	Assert.assertTrue("INSCRIPCION_CERRADA (subset 0-1)", model.updateSesiones(curso,
	sesiones.subList(0, 1), LocalDate.now()));

	LocalDate dateToTest = LocalDate.parse("2023-05-20");

	// EN_CURSO
	curso = getDTO("2");
	sesiones = model.getSesionesFromCurso(curso.getId());
	Assert.assertTrue("EN_CURSO (subset pasado)", model.updateSesiones(curso,
	sesiones.stream().filter(s ->
	LocalDate.parse(s.getFecha()).isBefore(dateToTest)).collect(Collectors.toList()),
	dateToTest));
	Assert.assertFalse("EN_CURSO (subset futuro)", model.updateSesiones(curso,
	sesiones.stream().filter(s ->
	LocalDate.parse(s.getFecha()).isAfter(dateToTest)).collect(Collectors.toList()),
	dateToTest));

	// FINALIZADO
	curso = getDTO("1");
	sesiones = model.getSesionesFromCurso(curso.getId());
	Assert.assertFalse("FINALIZADO (subset 0-1)", model.updateSesiones(curso,
	sesiones.stream().filter(s ->
	LocalDate.parse(s.getFecha()).isBefore(dateToTest)).collect(Collectors.toList()),
	LocalDate.now()));
}


@Test
public void testCostes() {
	/*
	* Los costes de un curso se pueden modificar:
	*
	* - Si el curso no tiene inscripciones, todo.
	* - Si el curso tiene inscripciones o está cerrado, nada.
	*/

	Assert.assertTrue("Sin inscripciones", model.updateCostes(getDTO("6"), new
	LinkedList<>()));
	Assert.assertFalse("Con inscripciones", model.updateCostes(getDTO("2"), new
	LinkedList<>()));
}
```

```
@Test
public void testDocencias() {
	/*
	* Las docencias de un curso se pueden modificar:
	*
	* - Si el curso no está en progreso, todo.
	* - Si el curso está en progreso, solo añadir nuevas docencias.
	* - Si el curso está finalizado, solo modificar acuerdos ya existentes.
	* - Si el curso está cerrado, nada.
	*/

	CursoDTO curso;
	List<DocenciaDTO> docencias;

	// PLANEADO
	curso = getDTO("5");
	Assert.assertTrue("PLANEADO (todo)", model.updateDocencias(curso, new LinkedList<>()));

	// EN_INSCRIPCION
	curso = getDTO("4");
	Assert.assertTrue("EN_INSCRIPCION (todo)", model.updateDocencias(curso, new
	LinkedList<>()));

	// INSCRIPCION_CERRADA
	curso = getDTO("3");
	Assert.assertTrue("INSCRIPCION_CERRADA (todo)", model.updateDocencias(curso, new
	LinkedList<>()));

	// EN_CURSO
	curso = getDTO("2");
	docencias = model.getListaProfesores(curso.getId());
	docencias.add(new DocenciaDTO(curso.getId(), "3", "100"));
	Assert.assertTrue("EN_CURSO (añadir)", model.updateDocencias(curso, docencias));
	Assert.assertFalse("EN_CURSO (eliminar)", model.updateDocencias(curso,
	docencias.subList(0, 1)));
	docencias.get(0).setRemuneracion("30");
	Assert.assertFalse("EN_CURSO (modificar)", model.updateDocencias(curso, docencias));

	// FINALIZADO
	curso = getDTO("1");
	docencias = model.getListaProfesores(curso.getId());
	docencias.add(new DocenciaDTO(curso.getId(), "3", "100"));
	Assert.assertFalse("FINALIZADO (añadir)", model.updateDocencias(curso, docencias));
	Assert.assertFalse("FINALIZADO (eliminar)", model.updateDocencias(curso,
	docencias.subList(0, 1)));
	docencias = model.getListaProfesores(curso.getId());
	docencias.get(0).setRemuneracion("30");
	Assert.assertTrue("FINALIZADO (modificar)", model.updateDocencias(curso, docencias));
}


}
```

4.8.3. Condiciones de ejecución

Las pruebas se ejecutan sobre una base de datos de prueba creada a partir de datos autogenerados con la herramienta de generación de bases de datos del proyecto, escrita en Ruby. Se han seleccionado los datos mínimos necesarios para todas las pruebas del código fuente anterior. El código del script que genera la base de datos se puede encontrar en el punto 1 del Anexo.

4.8.4. Valores esperados y resultados obtenidos

Todas las pruebas retornan correctamente, sin excepciones. Se comprueban absolutamente todas las condiciones descritas en la jerarquía del diseño de las pruebas.

4.9. Inscripciones múltiples

4.9.1. Diseño de las pruebas

A la hora de probar la inserción de inscripciones múltiples, se ha de probar en dos sitios a la vez: tanto en *InscripcionesMultiples* como en *InscripcionesMultiplesEntidad*. Pese a que son dos HU separadas, las condiciones deberían aplicar a ambas por igual a la hora de registrar nuevas inscripciones.

Las condiciones a comprobar son las siguientes:

- Que los datos compartidos de las inscripciones se mantienen (entidades a los que están asociados)
- Que no se permite la inserción cuando se trata de insertar más alumnos que las plazas libres.

4.9.2. Implementación del código

```
package g41.si2022.coiipa.inscripciones_multiples;

import java.time.LocalDate;
import java.util.LinkedList;
import java.util.List;

import org.junit.*;

import g41.si2022.coiipa.inscribir_multiples_usuarios.InscribirMultiplesUsuariosModel;
import
g41.si2022.coiipa.inscribir_multiples_usuarios_entidad.InscribirMultiplesUsuariosEntidadModel;
import g41.si2022.dto.AlumnoDTO;
import g41.si2022.util.db.Database;


public class InscripcionesMultiplesTest extends g41.si2022.coiipa.TestCase {


private static InscribirMultiplesUsuariosModel model1;
private static InscribirMultiplesUsuariosEntidadModel model2;


@Override
public void loadData() {
	try {
	this.getDatabase().executeScript(Database.SQL_LOAD);
	} catch (Exception e) {
		e.printStackTrace();
		System.exit(1);
	}
}


@BeforeClass
public static void initialize() {
	model1 = new InscribirMultiplesUsuariosModel();
	model2 = new InscribirMultiplesUsuariosEntidadModel();
}
```

```
private int getNumInscripciones(String cursoId) {
	return Integer.parseInt(getDatabase().executeQuerySingle("SELECT COUNT(*) FROM inscripcion
	WHERE curso_id = " + cursoId).toString());
}


/*
* Ambos model deberían funcionar igual en las funciones que compartan.
*/


@Test
public void testPrincipal() {
	List<AlumnoDTO> alumnos = new LinkedList<>();
	alumnos.add(new AlumnoDTO());
	alumnos.add(new AlumnoDTO());


	// Datos de ejemplo
	alumnos.get(0).setNombre("Alejandro");
	alumnos.get(0).setApellidos("Gallego");
	alumnos.get(0).setEmail("alejandrogallegogaimer@gmail.com");
	alumnos.get(0).setNombreColectivo("Estudiantes");


	alumnos.get(1).setNombre("Francisco Gabriel");
	alumnos.get(1).setApellidos("Puga Lojo");
	alumnos.get(1).setEmail("franciscopuga@protonmail.com");
	alumnos.get(1).setNombreColectivo("Estudiantes");


	// Prueba de registro de alumnos
	model1.insertAlumnos(alumnos);
	model2.insertAlumnos(alumnos);


	// Prueba de inscripción de alumnos
	int size = getNumInscripciones("4");
	model1.insertInscripciones(alumnos, LocalDate.now().toString(), "4");
	Assert.assertEquals(size + 2, getNumInscripciones("4"));


	loadData();


	size = getNumInscripciones("4");
	model2.insertInscripciones(LocalDate.now().toString(), "4", alumnos, "1");
	Assert.assertEquals(size + 2, getNumInscripciones("4"));
}



}
```

4.9.3. Condiciones de ejecución

Al igual que con el apartado anterior, las pruebas se ejecutan sobre una base de datos seleccionada manualmente sobre un conjunto de datos autogenerados por la herramienta del proyecto. El script que genera la base de datos se encuentra en el primer punto del Anexo.

4.9.4. Valores esperados y resultados de las pruebas

Todas las pruebas se ejecutan correctamente, aunque no se comprueba el resultado de las inserciones en la base de datos debido a la estructura del código del Model original. Puesto que son partes del código muy complejas de las que dependen otras partes del código, no es posible rescribir estas porciones del código, por lo que tan solo se realizan las pruebas que se puedan ejecutar sin modificar el código extensamente.

4.10. Insertar un pago de un alumno:

4.10.1. Objetivo

Insertar un pago de alumno, para confirmar su inscripción en un curso.

4.10.2. Descripción

Un alumno ha realizado un pago de un curso. Este pago ha llegado a la cuenta bancaria del COIIPA, mediante una transferencia y la secretaria, se dispone a registrarlo.

- Introducción del pago en el sistema, para una inscripción.
    a. Se verifica que la inscripción exista. En caso contrario, el proceso es abortado
    b. Si la inscripción existe, se procede al siguiente punto.
- Se verifica el importe del pago
    a. Si este importe, corresponde con el que le toca a la inscripción, se continúa adelante con el proceso de las inscripciones.
    b. Si es menor, el modelo devuelve un código de error.
    c. Si es mayor, el modelo también debería devolver un código de error, distinto del otro.
- Se verifica la fecha del pago
    a. Si es igual o inferior a la fecha de fin de inscripciones, el proceso se completa de forma satisfactoria.
    b. Si es superior, el proceso es cancelado, y el modelo devuelve un mensaje de error.

4.10.3. Diseño de las pruebas

Para cada caso de pruebas, se procede al borrado de todos los datos contenidos en la base de datos. A continuación, se inserta una inscripción con ID 1, además de un curso con ID 1, y fecha de fin de inscripciones el 2 de mayo de 2023.

Primera prueba unitaria **(testIDValidaNoValida)**. Se procede a probar a registrar un pago, para una inscripción existente y una no existente. Se prueba de la siguiente manera:

- Se procede a inscribir un pago, con un importe correcto, en la inscripción de ID 1 (existente). El sistema devuelve 0, que significa pago introducido con éxito.
- Se procede a inscribir un pago, con un importe correcto, en la inscripción con ID 2 (no existente). El sistema procede a devolver -1, indicando error, y a abortar el proceso de introducción del pago.

Finalmente, en la BBDD, se puede apreciar que únicamente se encuentra inscrito el pago correcto.

Segunda prueba unitaria **(testPagoAlumno)**. En esta prueba, se probará a introducir un pago, con distintos importes. Se realiza de la siguiente manera:

- Se introduce un pago para la inscripción con ID 1, con un importe de 50€, siendo este menor que el importe que se debería pagar por el curso. El sistema devuelve un 1, indicando que el pago es menor a la cantidad requerida y rechaza el pago.
- Se introduce un pago para la inscripción con ID 1, con un importe de 100€, siendo este igual que el importe que se debería pagar por el curso. El sistema devuelve un 0, indicando que el pago es igual a la cantidad requerida y registra el pago.
- Se introduce un pago para la inscripción con ID 1, con un importe de 150€, siendo este mayor que el importe que se debería pagar por el curso. El sistema devuelve un 2, indicando que el pago es mayor a la cantidad requerida y rechaza el pago.

Finalmente, en la base de datos, se puede apreciar que únicamente ha sido aceptado el pago con importe de 100€.

Tercera prueba unitaria **(testPagoFechasAlumno)**. Se prueba la clase de equivalencia de las fechas de entrada. Para ello se prueban distintos valores de entrada.

- Se introduce un pago con fecha menor a la de fin de inscripciones. El programa acepta el pago y lo registra.
- Se introduce un pago con fecha igual a la de fin de inscripciones. El programa acepta el pago y lo registra.
- Se introduce un pago con fecha superior a la del fin de las inscripciones. El programa devuelve un error y rechaza el pago.

Tras ejecutar cada entrada, se procede a revisar la base de datos. En las dos primeras entradas, el pago ha sido aceptado, mientras en la tercera entrada, el pago ha sido rechazado.

4.10.4. Implementación de las pruebas con Junit

```
@Test
public void testIDValidaNoValida() {
    GestionarInscripcionesModel gIM = new GestionarInscripcionesModel();
    regenerateBD(); //Regenero la BBDD
      assertEquals(0, gIM.registrarPago("100", "2023-05-01", "1", "1")); //Pago por importe
correcto, y id inscripcioón válida
     assertEquals(-1, gIM.registrarPago("100", "2023-05-01", "2", "1")); //Pago por importe
correcto, pero id inscripción no válida.

    //Obtengo los pagos de nuevo. Veo que sólo se ha inscrito el pago correcto.
    List<PagoDTO> listaPagos = gIM.allPagos();
       assertEquals("[PagoDTO(id=null, importe=100.0, importedevuelto=null, fecha=2023-05-01,
inscripcion_id=1)]", listaPagos.toString());
}

@Test
public void testPagoAlumno() {

    String importe1 = "50"; //Pago menos al importe del curso
    String importe2 = "100"; //Pago por un importe igua al importe del curso.
    String importe3 = "150"; //Pago por un importe superior al del curso.
    GestionarInscripcionesModel gIM = new GestionarInscripcionesModel();
    regenerateBD(); //Regenero la BBDD
    //Inserto varios pagos para probar. Todos al ID 1, que es la inscripción que existe.

     assertEquals(1, gIM.registrarPago(importe1, "2023-05-01", "1", "1")); //Pago por importe
inferior.
    assertEquals(0, gIM.registrarPago(importe2, "2023-05-01", "1", "1")); //Pago correcto
     assertEquals(2, gIM.registrarPago(importe3, "2023-05-01", "1", "1")); //Pago por importe
superior

    List<PagoDTO> listaPagos = gIM.allPagos(); //Obtengo los pagos-
       assertEquals("[PagoDTO(id=null, importe=100.0, importedevuelto=null, fecha=2023-05-01,
inscripcion_id=1)]", listaPagos.toString()); //Sólo hay insertado un pago.

}

@Test
public void testPagoFechasAlumno() {

    String importe1 = "100"; //Pago por un importe igua al importe del curso.
    GestionarInscripcionesModel gIM = new GestionarInscripcionesModel();
    regenerateBD(); //Regenero la BBDD
    //Inserto varios pagos para probar. Todos al ID 1, que es la inscripción que existe.
    assertEquals(0, gIM.registrarPago(importe1, "2023-05-01", "1", "1"));
    List<PagoDTO> listaPagos = gIM.allPagos();
       assertEquals("[PagoDTO(id=null, importe=100.0, importedevuelto=null, fecha=2023-05-01,
inscripcion_id=1)]", listaPagos.toString());

    regenerateBD(); //Regenero la BBDD
    assertEquals(0, gIM.registrarPago(importe1, "2023-05-02", "1", "1"));
    listaPagos = gIM.allPagos();
       assertEquals("[PagoDTO(id=null, importe=100.0, importedevuelto=null, fecha=2023-05-02,
inscripcion_id=1)]", listaPagos.toString());

    regenerateBD(); //Regenero la BBDD
    assertEquals(-1, gIM.registrarPago(importe1, "2023-05-03", "1", "1"));
    listaPagos = gIM.allPagos();
    assertEquals("[]", listaPagos.toString()); //Este pago no se debe insertar


}
```

4.11. Facturación

Objetivo: La secretaria busca registrar una factura.

Descripción:

1. Se selecciona el curso a facturar.
2. Se selecciona el docente al que se le factura.
3. Se introduce la fecha de emisión de la factura.
4. Se introduce el coste total de la factura.
    a. Éste debe coincidir con lo calculado por el sistema, en caso contrario, no será posible emitir la factura.

4.11.1. Profesores

4.11.1.1. Estado inicial de la BBDD

- La BBDD contiene el curso utilizado en 4.2.1.2.1.
- La BBDD contiene un profesor:
    - Nombre: "Pedro"
    - Apellidos: "Angel Martinez"
    - DNI: "545232121A"
    - Dirección: "C/ Albeniz, 5"
    - Email: panma@gmail.com
    - Teléfono: "664225423"
- El profesor Pedro está asociado con el curso de Kafka. Pedro imparte este curso por 250€.

4.11.1.2. Función a ejecutar

La función a probar será **`insertFactura(String fechaEmision, String docenciaId, String cantidad)`**.

Esta función retorna **`true`** o **`false`** en función de si ha sido posible o no insertar la información en la BBDD.

4.11.1.3. Pruebas realizadas

4.11.1.3.1. Factura correcta

```
@Test
/**
 * This test case will try to create a new Factura for a given docencia.
 * The costs match, so the entry will be created and a {@code true} will be returned.
 */
public void test1 () {
	assertEquals(
		true,
		this.gfpm.insertFactura("2023-05-01", this.docenciaId, "250")
	);
}
```

4.11.1.3.2. Factura incorrecta

```
@Test
/**
 * This test case will try to create a new Factura for a given docencia.
 * The costs will not match, so an error will be returned in the form of {@code false}.
 */
public void test2 () {
	assertEquals(
		false,
		this.gfpm.insertFactura("2023-05-01", this.docenciaId, "150")
	);
}
```

4.11.2. Empresas

4.11.1.1. Estado inicial de la BBDD

- La BBDD contiene el curso utilizado en 4.2.1.2.1.
- La BBDD contiene una empresa:
    - Nombre: Cursados
    - Email: info@cursados.com
    - Teléfono: 775221330
- La empresa Cursados está asociada con el curso Kafka. Cursados imparte el curso por 250€.

4.11.1.2. Función a ejecutar

La función a probar será **`insertFacturaEmpresa(String fechaEmision, String cursoId, String empresaId, String cantidad)`**.

Esta función retorna **`true`** o **`false`** en función de si ha sido posible o no insertar la información en la BBDD.

4.11.1.3. Pruebas realizadas

4.11.1.3.1. Factura correcta

```
@Test
/**
 * This test case will try to create a new Factura for a given curso.
 * The costs match, so the entry will be created and a {@code true} will be returned.
 */
public void test3 () {
	assertEquals(
		true,
		this.gfem.insertFacturaEmpresa("2023-05-01", this.cursoEmprId, this.emprId, "250")
	);
}
```

4.11.1.3.2. Factura incorrecta

```
@Test
/**
 * This test case will try to create a new Factura for a given curso.
 * The costs will not match, so the entry will not be created and a {@code false} will be
returned.
 */
public void test4 () {
	assertEquals(
		true,
		this.gfem.insertFacturaEmpresa("2023-05-01", this.cursoEmprId, this.emprId, "150")
	);
}
```

5. Anexo

5.1. Código de generación SQL (apartados 4.8 y 4.9)

```
INSERT INTO curso (nombre, descripcion, start_inscr, end_inscr, plazas, start, end, entidad_id, importe) VALUES
('[G] Curso ya finalizado', 'Curso generado que ya ha finalizado', '2023-03-01', '2023-03-31', '17', '2023-04-01', '2023-04-30', '', ''),
('[G] Curso en progreso', 'Curso generado que está en progreso', '2023-04-01', '2023-04-30', '152', '2023-05-01', '2023-05-31', '', ''),
('[G] Curso con inscripción cerrada', 'Curso generado que ya ha cerrado la inscripción pero no ha empezado', '2023-04-01', '2023-04-30', '203', '2023-06-01', '2023-06-30', '', ''),
('[G] Curso con inscripción abierta', 'Curso generado que está abierto a inscripciones', '2023-05-01', '2023-05-31', '134', '2023-06-01', '2023-06-30', '', ''),
('[G] Curso esperando a inscripción', 'Curso generado que está esperando a que se abra la inscripción', '2023-06-01', '2023-06-30', '133', '2023-03-01', '2023-03-31', '', ''),
('[G] 2 sin inscripciones', 'ayuda por favor', '2023-04-01', '2023-04-30', '152', '2023-05-01', '2023-05-31', '', '');


INSERT INTO sesion (loc, fecha, hora_ini, hora_fin, curso_id, observaciones) VALUES
('AN-B3', '2023-04-04', '12:00:00', '15:00:00', '1', 'Sesion generado automáticamente'),
('AN-S6', '2023-04-06', '15:00:00', '16:00:00', '1', 'Sesion generado automáticamente'),
('DO-1.S.31', '2023-04-08', '08:00:00', '09:00:00', '1', 'Sesion generado automáticamente'),
('AN-E', '2023-05-18', '09:00:00', '10:00:00', '2', 'Sesion generado automáticamente'),
('AN-B1', '2023-05-29', '15:00:00', '18:00:00', '2', 'Sesion generado automáticamente'),
('AN-B6', '2023-05-28', '09:00:00', '11:00:00', '2', 'Sesion generado automáticamente'),
('DO-9', '2023-06-13', '10:00:00', '11:00:00', '3', 'Sesion generado automáticamente'),
('AN-B3', '2023-06-05', '08:00:00', '10:00:00', '3', 'Sesion generado automáticamente'),
('AN-C', '2023-06-16', '15:00:00', '17:00:00', '3', 'Sesion generado automáticamente'),
('AN-E', '2023-06-05', '13:00:00', '15:00:00', '3', 'Sesion generado automáticamente'),
('AN-B', '2023-06-15', '09:00:00', '11:00:00', '3', 'Sesion generado automáticamente'),
('DO-9', '2023-06-13', '13:00:00', '16:00:00', '4', 'Sesion generado automáticamente'),
('AN-S6', '2023-06-24', '13:00:00', '15:00:00', '4', 'Sesion generado automáticamente'),
('AN-C', '2023-06-20', '12:00:00', '15:00:00', '4', 'Sesion generado automáticamente'),
('AN-B', '2023-03-28', '13:00:00', '14:00:00', '5', 'Sesion generado automáticamente'),
('DO-1.S.31', '2023-03-19', '10:00:00', '12:00:00', '5', 'Sesion generado automáticamente'),
('DO-1.S.31', '2023-03-01', '16:00:00', '18:00:00', '5', 'Sesion generado automáticamente'),
('AS-1', '2023-03-10', '15:00:00', '18:00:00', '5', 'Sesion generado automáticamente'),
('DO-1.S.31', '2023-03-23', '14:00:00', '15:00:00', '5', 'Sesion generado automáticamente'),
('AN-E', '2023-05-18', '09:00:00', '10:00:00', '6', 'Sesion generado automáticamente'),
('AN-B1', '2023-05-29', '15:00:00', '18:00:00', '6', 'Sesion generado automáticamente'),
('AN-B6', '2023-05-28', '09:00:00', '11:00:00', '6', 'Sesion generado automáticamente');


INSERT INTO docente (nombre, apellidos, email, telefono, direccion, dni) VALUES
('[G] Arturo', 'Aguayo', 'sharyl.moen@witting.co', '666949598', 'Edificio Carla Cadena 81 Esc. 584', '67102814-S'),
('[G] Micaela', 'Montez', 'bennett.bashirian@beatty.info', '659775869', 'Escalinata Marisol 14 Esc. 207', '26704311-T'),
('[G] Irene', 'Arreola', 'byron.brekke@batz.name', '665212965', 'Urbanización Jaime Abreu 5', '71998968-K'),
('[G] Felipe', 'Otero', 'brain.johns@quitzon.name', '610376914', 'Prolongación Antonia Jimínez, 13', '67477296-B'),
('[G] Elisa', 'Echevarría', 'dante.wuckert@kuphal.io', '677498064', 'Camino Florencia 99', '45575812-R');
```

```
INSERT INTO docencia (curso_id, docente_id, remuneracion) VALUES
('1', '1', '209'),
('1', '2', '217'),
('1', '3', '139'),
('2', '1', '269'),
('2', '2', '452'),
('2', '3', '362'),
('2', '4', '112'),
('3', '1', '460'),
('4', '2', '178'),
('5', '4', '327');

INSERT INTO coste (curso_id, colectivo_id, coste) VALUES
('1', '0', '39'),
('1', '1', '235'),
('1', '2', '243'),
('2', '0', '157'),
('2', '1', '11'),
('2', '2', '104'),
('3', '0', '142'),
('3', '1', '73'),
('3', '2', '214'),
('4', '0', '14'),
('4', '1', '229'),
('4', '2', '168'),
('5', '0', '215'),
('5', '1', '128'),
('5', '2', '63');

INSERT INTO colectivo (nombre) VALUES
('Estudiantes'),
('Colegiados'),
('Otros');

INSERT INTO inscripcion (curso_id, alumno_id, fecha, cancelada, entidad_id, coste_id) VALUES
('1', '1', '2023-04-02', '0', '', '1'),
('1', '2', '2023-04-02', '0', '', '1'),
('2', '1', '2023-05-27', '0', '', '5'),
('2', '2', '2023-05-27', '0', '', '5'),
('3', '1', '2023-06-11', '0', '', '8'),
('3', '2', '2023-06-11', '0', '', '8'),
('4', '1', '2023-06-19', '0', '', '10'),
('4', '2', '2023-06-19', '0', '', '10'),
('5', '1', '2023-03-20', '0', '', '13'),
('5', '2', '2023-03-20', '0', '', '13');

INSERT INTO alumno (nombre, apellidos, email, telefono) VALUES
('Juan', 'Mier', 'mier@mier.info', ''),
('Test', 'Test', 'test@test.com', '');
```

5.2. Código de generación SQL (apartado 4.3)

```
INSERT INTO curso (nombre, descripcion, start_inscr, end_inscr, plazas, start, end, estado, entidad_id, importe) VALUES
('Curso con estado EN_INSCRIPCION', 'Test', '2023-05-07', '2023-05-13', '5', '2023-05-15', '2023-05-20', '', '', ''),
('Curso con estado EN_CURSO', 'Test', '2023-04-20', '2023-04-25', '5', '2023-05-08', '2023-05-15', '', '', ''),
('Curso con estado PLANEADO', 'Test', '2023-07-07', '2023-07-13', '5', '2023-07-15', '2023-07-20', '', '', ''),
('Curso con estado FINALIZADO', 'Test', '2023-01-07', '2023-01-13', '5', '2023-01-15', '2023-01-20', '', '', ''),
('Curso con estado CANCELADO', 'Test', '2023-01-07', '2023-01-13', '5', '2023-01-15', '2023-01-20', 'CANCELADO', '', '');

INSERT INTO inscripcion (fecha, cancelada, curso_id, alumno_id, coste_id, entidad_id) VALUES
('2023-05-08', '0', '1', '1', '2', ''),
('2023-05-09', '0', '1', '2', '2', ''),
('2023-05-10', '1', '1', '3', '2', '');

INSERT INTO coste (coste, curso_id, colectivo_id) VALUES
('80', '1', '1'),
('75', '3', '2');

INSERT INTO colectivo (nombre) VALUES
('Por defecto 1'),
('Por defecto 2');

INSERT INTO pago (importe, fecha, inscripcion_id) VALUES
('80', '2023-05-08', '1', '');

INSERT INTO alumno (nombre, apellidos, email, telefono) VALUES
('Pedro', 'Benito', 'pedrobenito@gmail.com', '123456789'),
('María', 'Magdalena', 'mariamagdalena@gmail.com', '987654321'),
('Ahmad', 'Chacur', 'ahmadchacur@gmail.com', '789456123');
```

5.3. Código de generación SQL (apartado 4.4)

```
INSERT INTO inscripcion (fecha, cancelada, curso_id, alumno_id, coste_id) VALUES
('2023-05-08', '0', '1', '1', '1'),
('2023-04-22', '1', '2', '2', '2'),
('2023-04-23', '0', '2', '3', '2');

INSERT INTO curso (nombre, descripcion, start_inscr, end_inscr, plazas, start, end, estado, entidad_id, importe) VALUES
('Curso con estado EN_INSCRIPCION', 'Test', '2023-05-07', '2023-05-13', '5', '2023-05-15', '2023-05-20', '', '', ''),
('Curso con estado EN_CURSO', 'Test', '2023-04-20', '2023-04-25', '5', '2023-05-08', '2023-05-15', '', '', '');

INSERT INTO alumno (nombre, apellidos, email, telefono) VALUES
('Pedro', 'Benito', 'pedrobenito@gmail.com', '123456789'),
('María', 'Magdalena', 'mariamagdalena@gmail.com', '987654321'),
('Ahmad', 'Chacur', 'ahmadchacur@gmail.com', '789456123');

INSERT INTO coste (coste, curso_id, colectivo_id) VALUES
('80', '1', '1'),
('75', '3', '2');

INSERT INTO colectivo (nombre) VALUES
('Por defecto 1'),
('Por defecto 2');
```